

Introduction to Intel DPDK

Oct 24th, 2014

IGEL Co.,Ltd.

Tetsuya Mukawa

武川 哲也

はじめに



Q: Intel DPDK(以下、DPDK)って？

A: 高スループット/低レイテンシのネットワークを実現する仕組みです。

Q: DPDKの目的は？

A: 高価なNW機器と同等の機能・性能を、Linux/BSD上のソフトウェアで実現することです。

Q: DPDKは、サーバ用途の技術では？

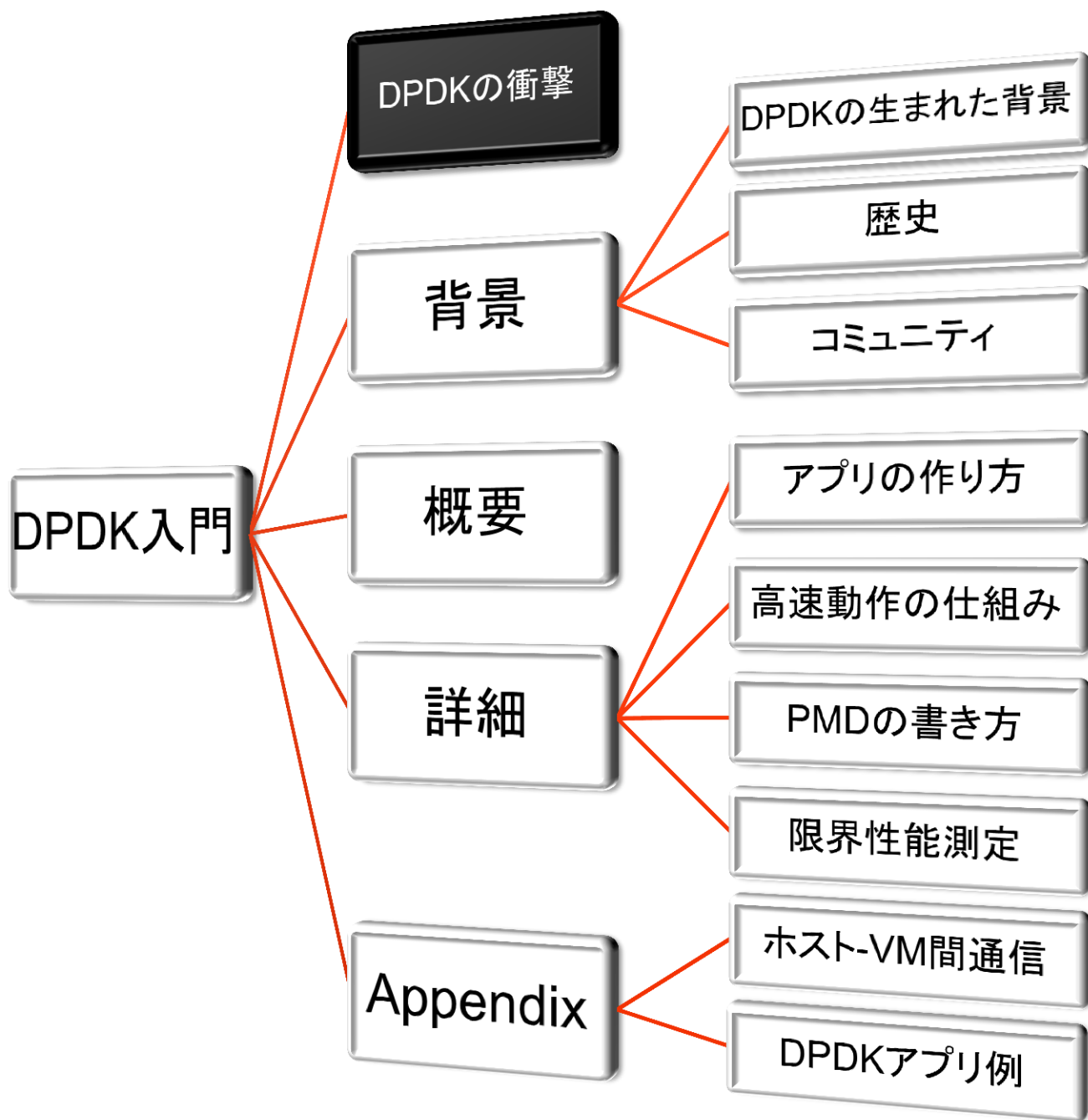
A: 今のところ、その通りです。

Q: なぜ、CELFで？

A: 高速化手法が面白いので紹介します。

Q: あなたは誰？

A: 組み込みエンジニアです。たまに、CELFに参加しています。



DPDKの衝撃① ～高速転送～

■ 公称、“Over 160Mpps(fps)”

- 64byte(ショート)パケットで、約80Gbps
- 1024byteパケットで、約1300Gbps

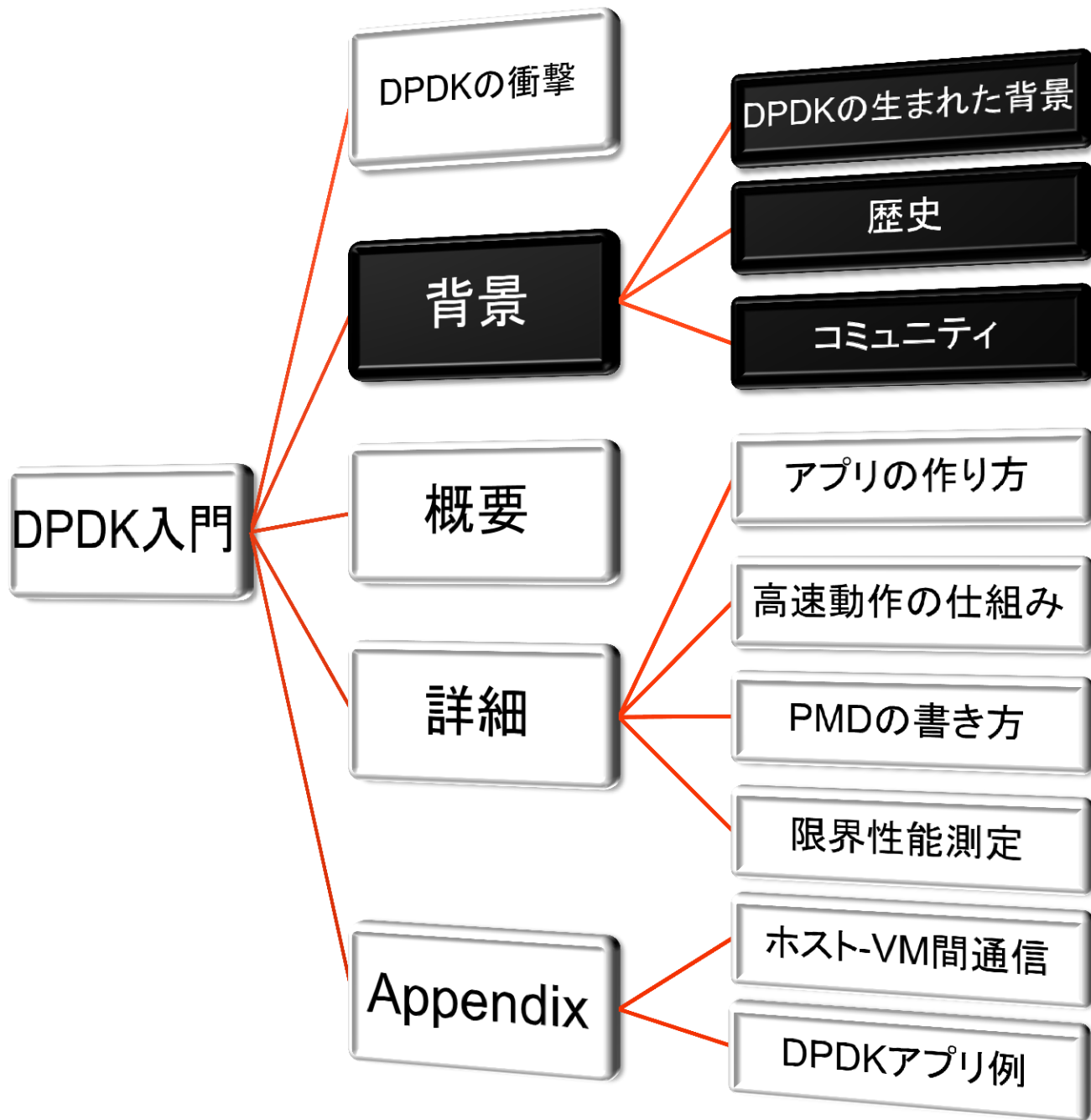
通信事業にとっては、
ショートパケットのパ
フォーマンスが重要らしい
(聞いた話)

現在手に入るNICの最速
は40Gbps(?)

- というわけで、160Mppsは、信じられないくらい速い。
 - 信じられないので、調べてみました(後述)。

DPDKの衝撃② ～安価～

- BSDライセンス
- Linux/BSD上で動作
- x86だけでなく、ARMやPower上でも動作
 - ATOMも
- 専用機器に比べて非常に安価なサーバ上で、専用機器と同等の性能・機能を持つネットワーク機器を作成することが可能に。
 - ネットワーク業界に大きなインパクト



DPDKの生まれた背景 ～想像です～

最近では...

Linaro

IBM

ARMポーティングは、Linaro主導
<https://wiki.linaro.org/LNG/Engineering/DPDK>

Powerポーティングは、IBM主導

【通信事業者】

NW専用機器は高すぎる

【Intel】

自社のCPUやNIC
を売りたい

【ソフトウェア会社】

寡占だったNW専用機器にビジネスチャンス

DPDK初期では...

6WIND

Wind River

DPDKの歴史

年月	バージョン	拡張された機能
2012年11月	DPDK-1.2.3	
2013年6月	DPDK-1.3.1	
2013年8月	DPDK-1.4.1	
2013年9月	DPDK-1.5.0	
2013年10月	DPDK-1.5.1	
2014年1月	DPDK-1.6.0	
2014年6月	DPDK-1.7.0	
2014年11月？	DPDK-1.8.0	

機能の詳細については後述

VMとの
通信拡張

数種の仮想
デバイスサ
ポート

アプリ制作に
便利なライブ
ラリの追加

比較的、新しい
ソフトウェア

最近になって、アプリ用のライブラリ
が充実してきている

DPDK-1.4?

Intelの対応

- Intelの公式サイトにて、ソースコードのみ公開
- gitは非公開

6WINDの対応

- dpdk.orgを立ち上げ、独自にMLとgitを提供
- Intelのソースから作成したgit tree

利用者の反応

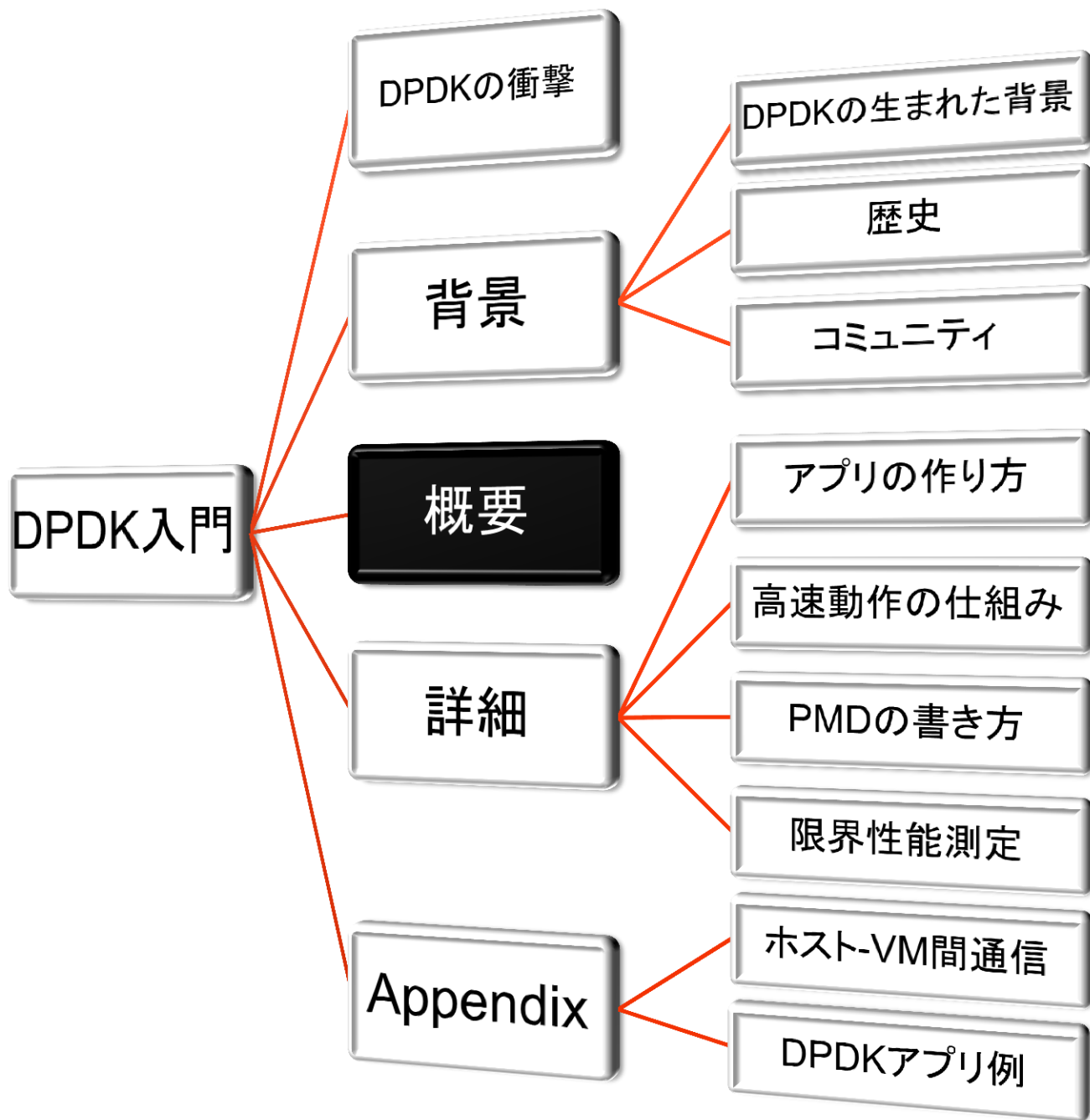
- dpdk.orgにpatchを投稿
- IntelのDPDK-1.5より、dpdk.orgのDPDK-1.5の方が高機能に。

DPDK-1.6

Intelの対応

- dpdk.orgを公式なgitと認定
- Intel自身も、dpdk.orgに投稿
- メンテナは、Intelではなく、6WIND

**コミュニティでは、Intelと共に
6WINDにも存在感**



- 例えばLinuxでは、1Mpps以下
 - 1500byte程度のパケットでも、実測10Gbpsを下回るらしい。
- 既存デバドラによる転送はなぜ遅いか？
 - 割り込みによるオーバーヘッド
 - コンテキストスイッチのオーバーヘッド
 - TLBミス
 - (CPU Coreの) キャッシュミス

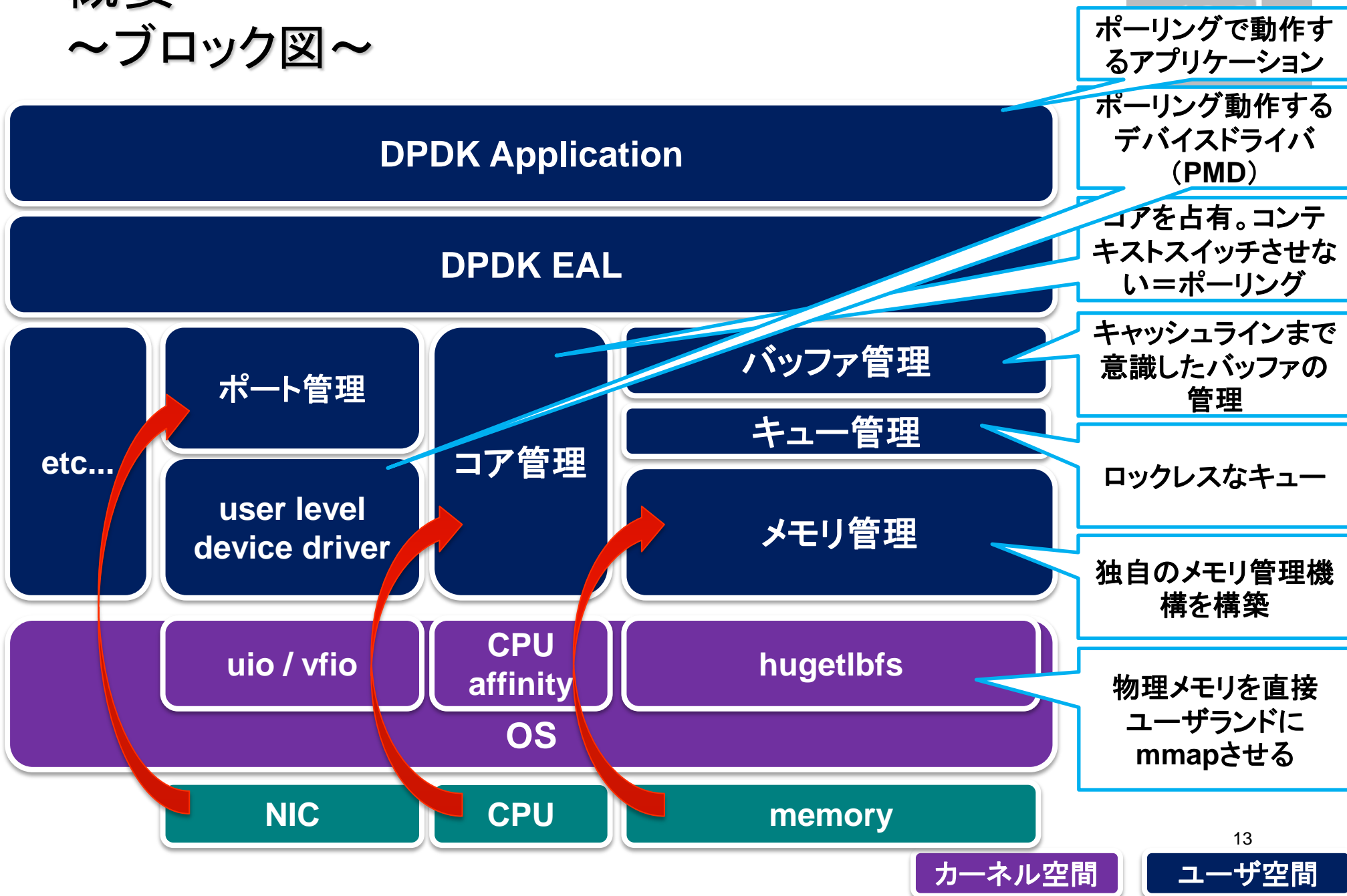
概要

～DPDKのアーキテクチャ(超基本)～

サーバ対象のソフトウェアなのに、
uio/vfioと組みみたい

- uio/vfioを使ったユーザースペースデバイスドライバとライブラリ
 - (主に)IntelのNIC対象
- ユーザ空間から独自のリソース管理
 - コア
 - メモリ
 - 独自のメモリ管理の仕組みを構築

概要 ～ブロック図～



概要

～実際の操作例①～

■ 準備（DPDKはコンパイルしていること前提）

– hugepageをmount

- Bootオプションを以下のように変更

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash  
default_hugepagesz=1G hugepagesz=1G hugepages=10  
norandmaps=0"
```

- fstabに以下のように設定

```
nodev          /mnt/huge      hugetlbfs      pagesize=1GB  0  0
```

– おまじないを唱える

```
$ sudo modprobe uio  
$ sudo insmod ./build/kmod/igb_uio.ko
```

**./dpdkにてDPDKを
コンパイルしたと想定**

概要

～実際の操作例②～

■ サンプルアプリケーション(testpmd)の実行まで

– DPDKのuioドライバ配下に、NICを配置

- まずは、現状確認

```
Network devices using DPDK-compatible driver
=====
<none>
```

```
Network devices using kernel driver
=====
```

現在は、
e1000eドライバ配下

```
0000:02:00.0 '82572EI Gigabit Ethernet Controller
(Copper)' if=p4p1 drv=e1000e unused=
0000:06:00.0 'RTL8111/8168/8411 PCI Express Gigabit
Ethernet Controller' if=eth0 drv=r8169 unused= *Active*
```

```
Other network devices
=====
<none>
```

概要

～実際の操作例③～

■ サンプルアプリケーション(testpmd)の実行まで

– DPDKのuioドライバ配下に、NICを配置

- DPDKのuioドライバ(igb_uio)にbind

```
$ sudo ./dpdk/tools/dpdk_nic_bind.py -b igb_uio  
0000:02:00.0
```


概要

～実際の操作例④～

■ サンプルアプリケーション(testpmd)の実行まで

– DPDKのuioドライバ配下に、NICを配置

- igb_uioにbindされているか確認

```
Network devices using DPDK-compatible driver
=====
```

現在は、DPDK配下

```
0000:02:00.0 '82572EI Gigabit Ethernet Controller
(Copper)' if=p4p1 drv=e1000e unused=
```

```
Network devices using kernel driver
=====
```

```
0000:06:00.0 'RTL8111/8168/8411 PCI Express Gigabit
Ethernet Controller' if=eth0 drv=r8169 unused= *Active*
```

```
Other network devices
=====
```

```
<none>
```

概要

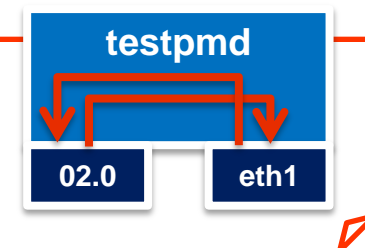
～実際の操作例⑤～

■ サンプルアプリケーション(testpmd)の実行まで

– testpmdを実行

```
$ sudo ./dpdk/build/app/testpmd -c f -n 1 -vdev  
"eth_pcap0,iface=eth1" -- -i  
(略)  
testpmd>
```

あとは、startコマンドにて、
0000:02:00.0とeth1間の
双方向データ転送開始



– オプションの意味

-c	このアプリケーションを動作させるコアマスク指定 f は、Core0～Core3にて、DPDKのスレッドが実行されることを示す。
-n	CPUソケットごとのメモリチャンネルの数
vdev	仮想デバイスの追加指定。この場合、ibpcapでeth1にアクセスする仮想デバイスをアタッチしている。
--	これ以前は、DPDK共通のオプション。これ以後は、アプリケーション固有のオプション
-i	インタラクティブモードで起動

概要

～まとめ①～

■ uio/vfioを使ったユーザースペースデバイスドライバとライブラリ

- (主に) IntelのNIC対象

本当に速いの？

■ ユーザ空間から独自のリソース管理

- コア
- メモリ
 - 独自のメモリ管理の仕組みを構築

どうやって？

概要

～まとめ②～

■ uio / vfioを使っても速い理由

- コアを占有してポーリング動作

■ どうやって、独自のリソース管理をするか

- コアを占有するために、既存のAffinity用のAPIを利用
 - コアを占有なので、管理といって良いかは微妙だが・・・
- hugetlbfsを利用して、物理メモリを直接マッピング
 - hugepage(1Mや1G)なので、TLBミスも大幅減
- 獲得したメモリ上に、独自のメモリ管理機構を構築
 - 詳細は後述

概要

～まとめ③～

igel

速さの秘密は、
ポーリング

キャッシュに載りやすい
コンテキストスイッチなし

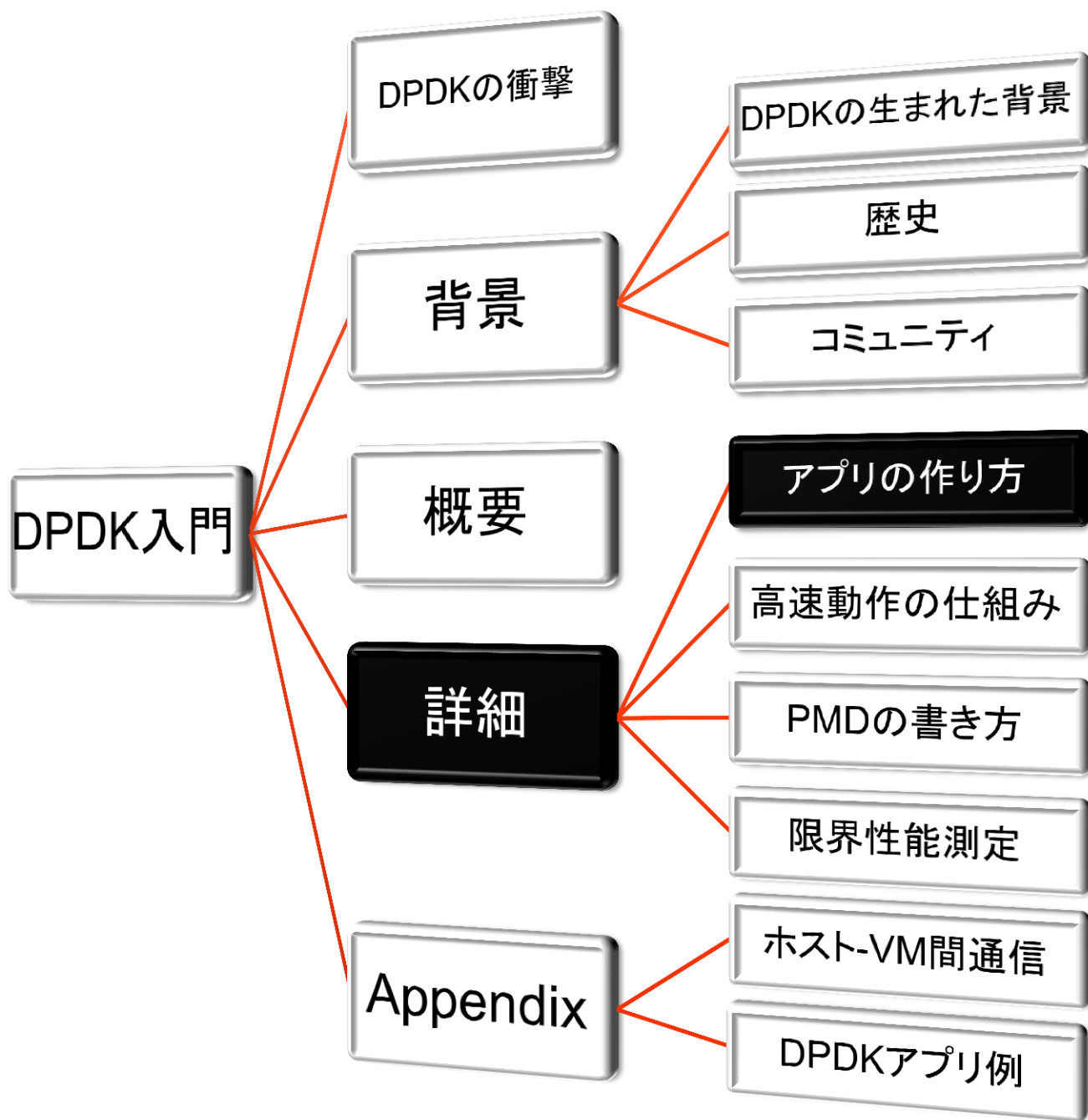
割り込み駆動の必要なし

ポーリング動作

効率よくポーリングさせる仕組みを
DPDKは提供

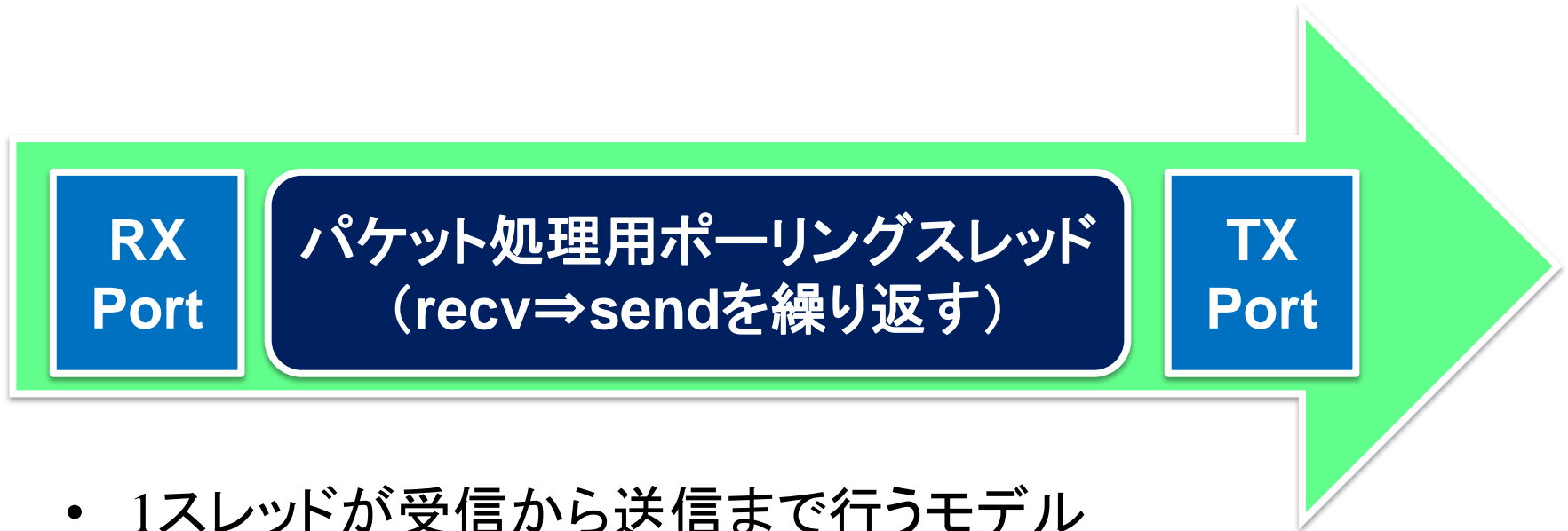
- ロックしない
- キャッシュミス減らす

実は、アプリ実装時にも意識する必要
がある。



詳細

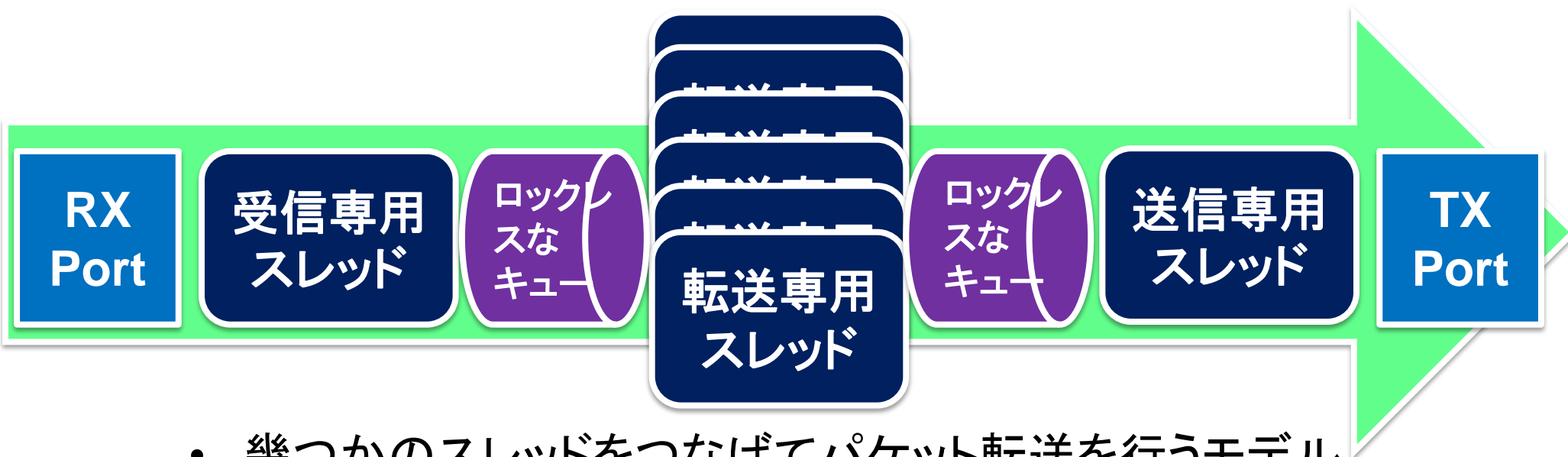
～アプリの作り方～ Run to completionモデル



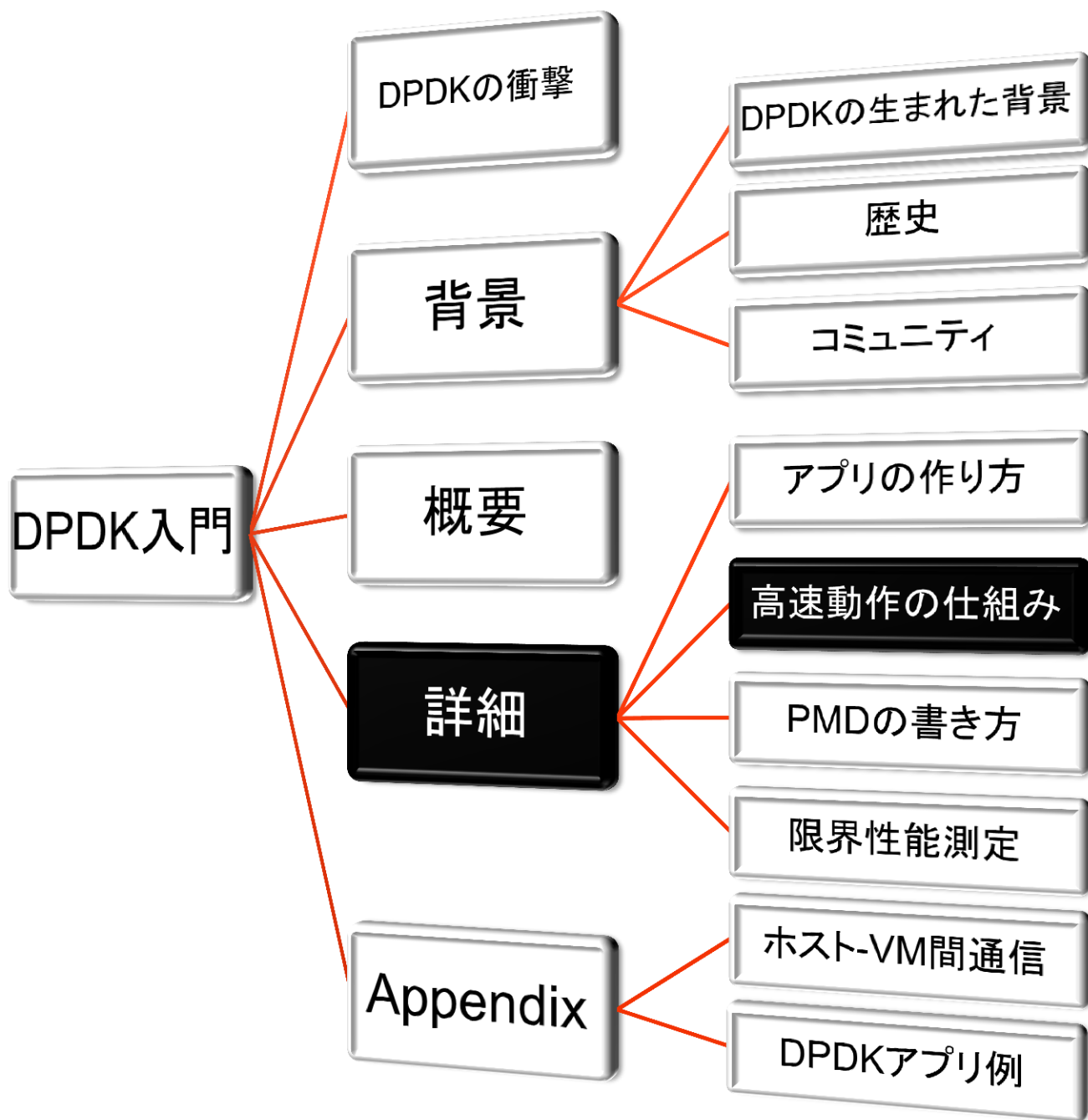
- 1スレッドが受信から送信まで行うモデル
 - 単純な処理ならこれで十分
 - レイテンシも抑えられる。
- なお、各ポートにアクセスできるのは1スレッドまで
 - 複数スレッドを使った並列化は不可能
 - この制約は、PMDの仕様による。

詳細

～アプリの作り方～ Pipelineモデル



- 幾つかのスレッドをつなげてパケット転送を行うモデル
 - “転送専用スレッド”で複雑なパケット操作を行う。
 - 当然、各スレッドごとにコアを1つ占有する。┐(´-`)┐
 - 複雑な処理を複数コアに分けて並列度をあげる。
 - 転送順を乱してはいけない場合は、何か考える必要あり。
- 複雑なアプリはだいたいこの方式の派生モデルで実装(?)
- 事前に処理能力を想定しておく必要あり。
 - 伝統的な組込みっぽい。



～高速動作の仕組み～ ロックしない仕組み①

■ Pipelineモデル等に使われるロックレスなキュー

- もともとLinux/BSDで使用されてきたキューの軽量版
 - スレッドがコアを占有することを前提にさらに高速化
- DPDKでは、“Ring”と呼ばれる。
 - 例えば、バッファのアドレスをキューイングする。
- Single/Multi Producer – Single/Multi Consumerに対応
 - キューに対して複数スレッドから同時にアクセス可能
 - ただし、Multi環境では、完了するまでロックしないがBusy waitすることがある。



Producer/Consumer共に、シングルでもマルチスレッドでも良い！

■ Ringを使用する際の制約

- multi producersにて、あるRingに対してenqueueするスレッドは、同一のRingに対してmulti producersでenqueueする別スレッドにpreemptされてはならない。
- multi consumersにて、あるRingに対してdequeueするスレッドは、同一のRingに対してmulti consumersでdequeueする別スレッドにpreemptされてはならない。
- DPDKでは、1スレッドでコアを占有するため、Ringの操作中にpreemptされることがない。

～高速動作の仕組み～ ロックしない仕組み②

■ ロックしない仕組みではないが...

- 各PMD(デバイスドライバ)は、同一ポートに対して複数スレッドがアクセスしてくることはないという前提で実装
- ポートにアクセス可能なスレッドが一つの理由

～高速動作の仕組み～ キャッシュの活用(TLB)

■ TLBミスの発生回数を低減させる仕組み

– hugetlbfsを利用

- DPDKから物理メモリをmapするためにも利用
- 2MBや1GBのpagesizeを利用
- 黙っていると、DPDKのアプリケーションは、利用可能な全てのhugetlbfs用のメモリをmapしようとします。⌋ (´-`) ⌋
 - オプションで使用メモリを指定可能

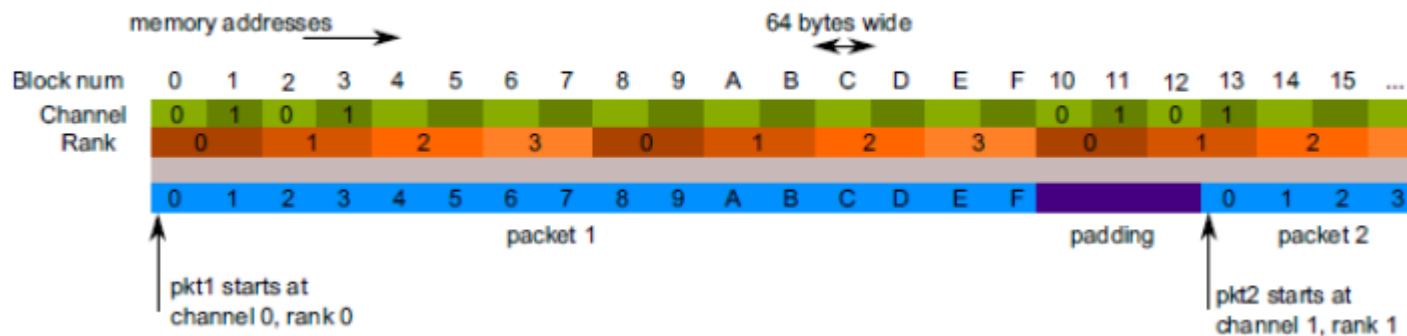
詳細

～高速動作の仕組み～ キャッシュの活用(Data)

■ DPDK独自のメモリ管理

- CPUキャッシュを効率的に使えるように、キャッシュのアライメント、DDR3やDIMMのチャンネル数、ランク数を意識してメモリを割り当てる仕組み
- ちなみに、NUMA構成も考慮

Two Channels and Quad-ranked DIMM Example



(DPDK Programers guideより)

～高速動作の仕組み～ コア単位のリサイクルキュー

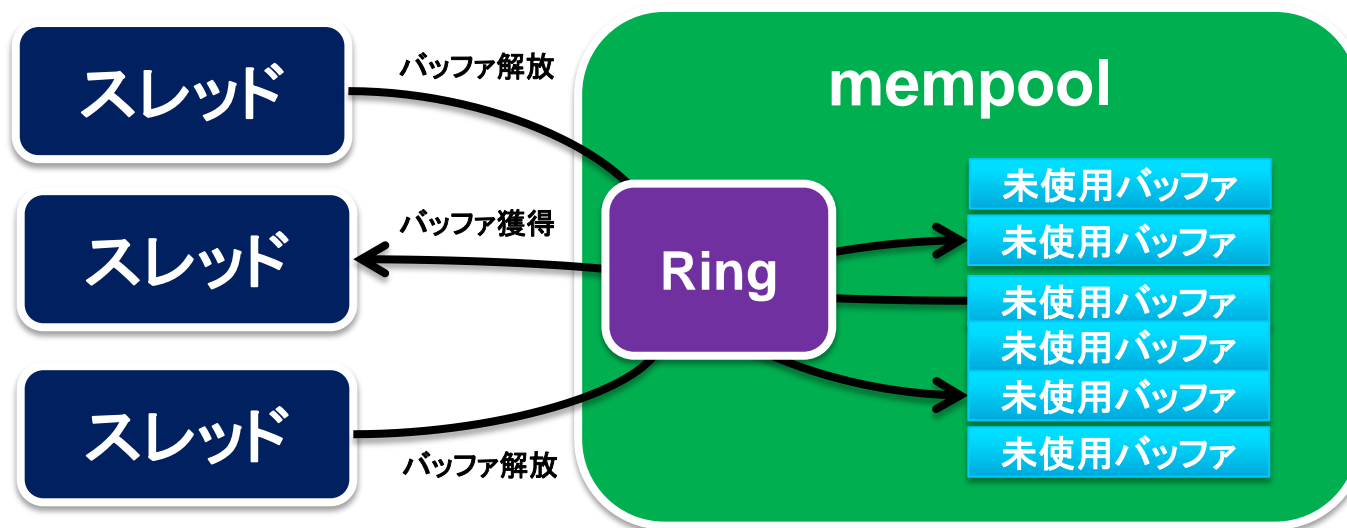
- コア別のバッファ・リサイクルキューを持つ。
 - DPDKは、マルチプロセス・マルチスレッド対応
 - 複数のタスクから共有されるデータは、hugetlbfs上に構築された**独自のメモリ機構の上**に置かれる。
 - …が、なるべく、一つのデータにアクセスしたくない。
 - パケットバッファについては、巧妙な仕組みで最適化

詳細

～高速動作の仕組み～ コア単位のデータ構造(contd.)

■ パケット用バッファには、巧妙な仕掛けがある。

- このバッファは、もちろん、キャッシュラインを考慮して獲得されたもの。
- バッファの集合(mempool)を用意し、mempoolの使用状況をRingで管理
- 複数スレッド(コア)からRingにアクセスし、ロックせずに高速にバッファを取得解放可能



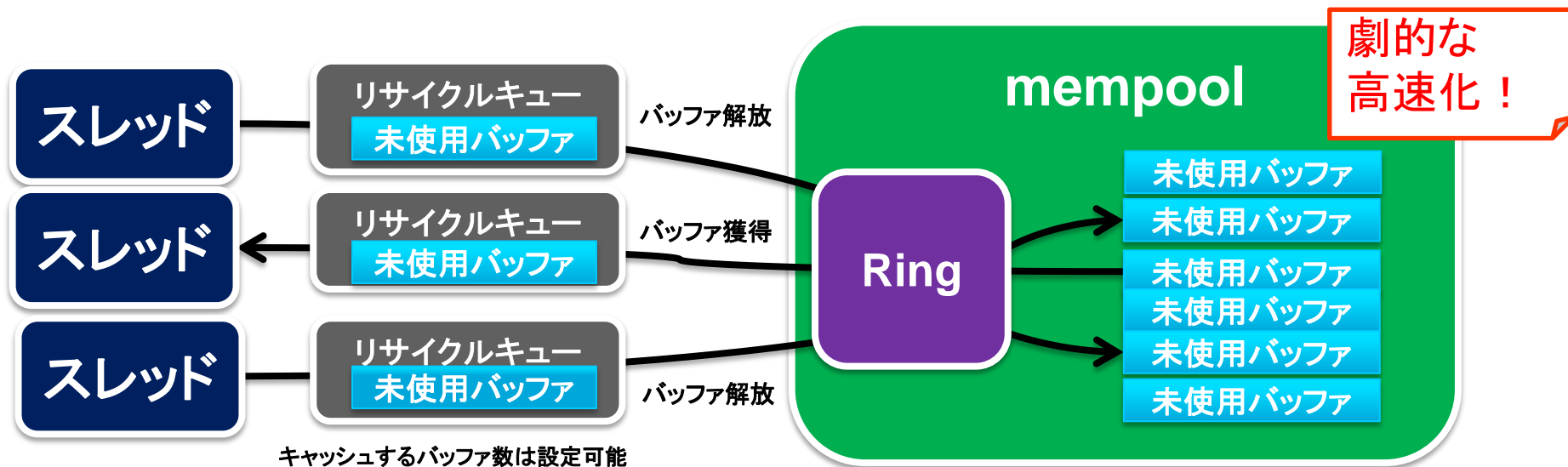
複数タスクからアクセスすると、RingはBusy wait することがあるんじゃない...???

この図は、実は正しくありません。次ページが正しい図

詳細

～高速動作の仕組み～ コア単位のデータ構造(contd.)

- 複数タスクが同一Ringにアクセスする場合、ロックはしないが、**Busy wait**することがある。
 - このオーバーヘッドも低減したい
- スレッドごとに、バッファのリサイクルキュー(実際は配列)持つ。
 - **使い終わっても直ぐに返しにはいかない。**
 - バッファ獲得要求に対し、Ringに返さずにとっておいたバッファを割り当てる。



詳細

～高速動作の仕組み①～ SSEの利用

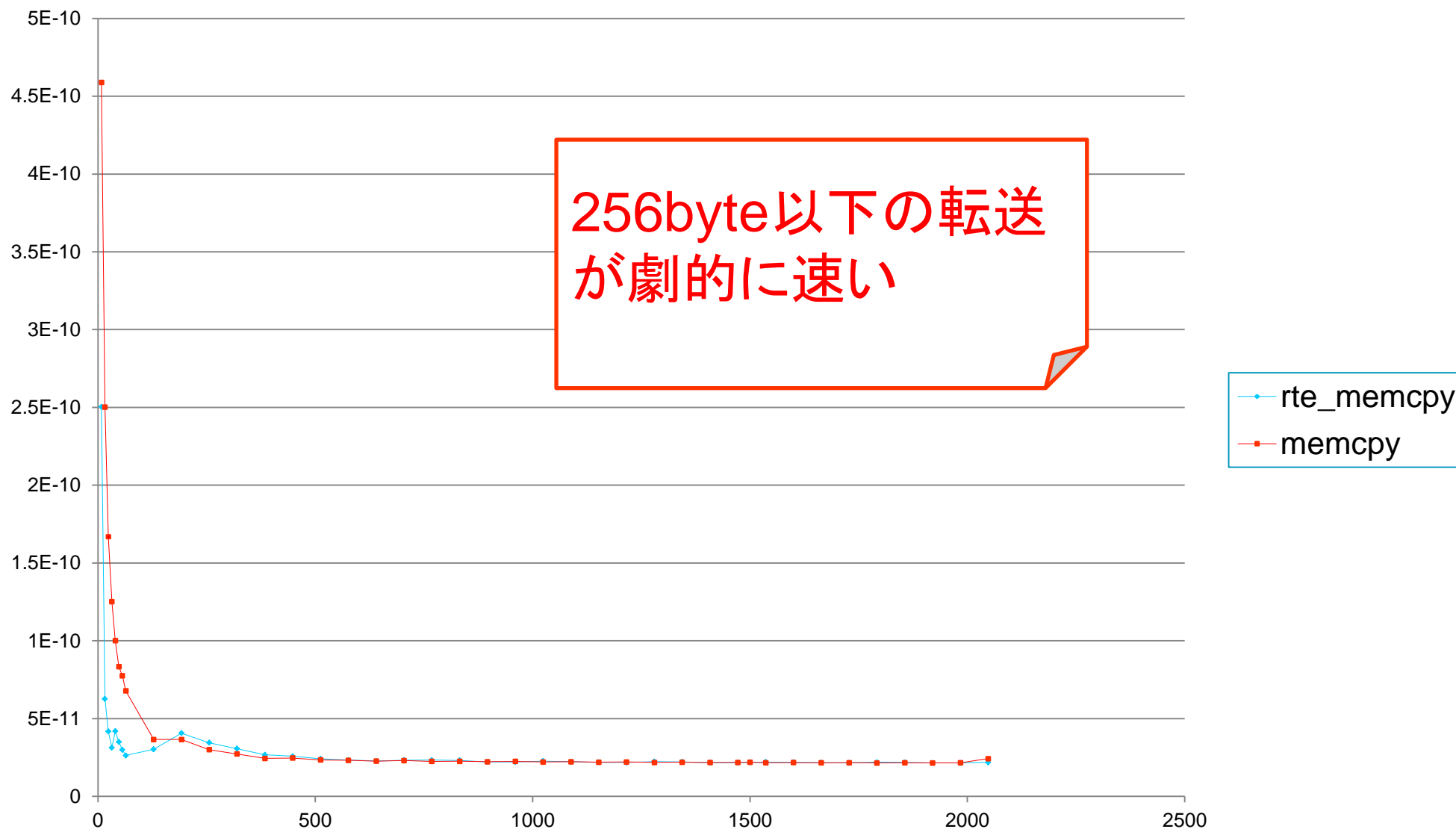


- DPDKにおけるSSE利用の一例
- DPDKは、独自のmemcpy APIを提供
 - rte_memcpy
- DPDKスレッド上で、DPDKのメモリ機構上のメモリをコピーする速度を比較

バッファサイズ	平均 rte_memcpy (sec/byte)	平均 memcpy (sec/byte)
8	2.50295E-10	4.58804E-10
16	6.25674E-11	2.50263E-10
24	4.1712E-11	1.66842E-10
32	3.12835E-11	1.25132E-10
40	4.19413E-11	1.00106E-10
48	3.49382E-11	8.34208E-11
56	2.99602E-11	7.74625E-11
64	2.6234E-11	6.77814E-11
128	3.02553E-11	3.65069E-11
192	4.06138E-11	3.65201E-11
256	3.44818E-11	2.99948E-11
320	3.06291E-11	2.72697E-11
384	2.66848E-11	2.44344E-11
448	2.58069E-11	2.46057E-11
512	2.41141E-11	2.34078E-11

詳細

～高速動作の仕組み②～ SSE拡張の利用

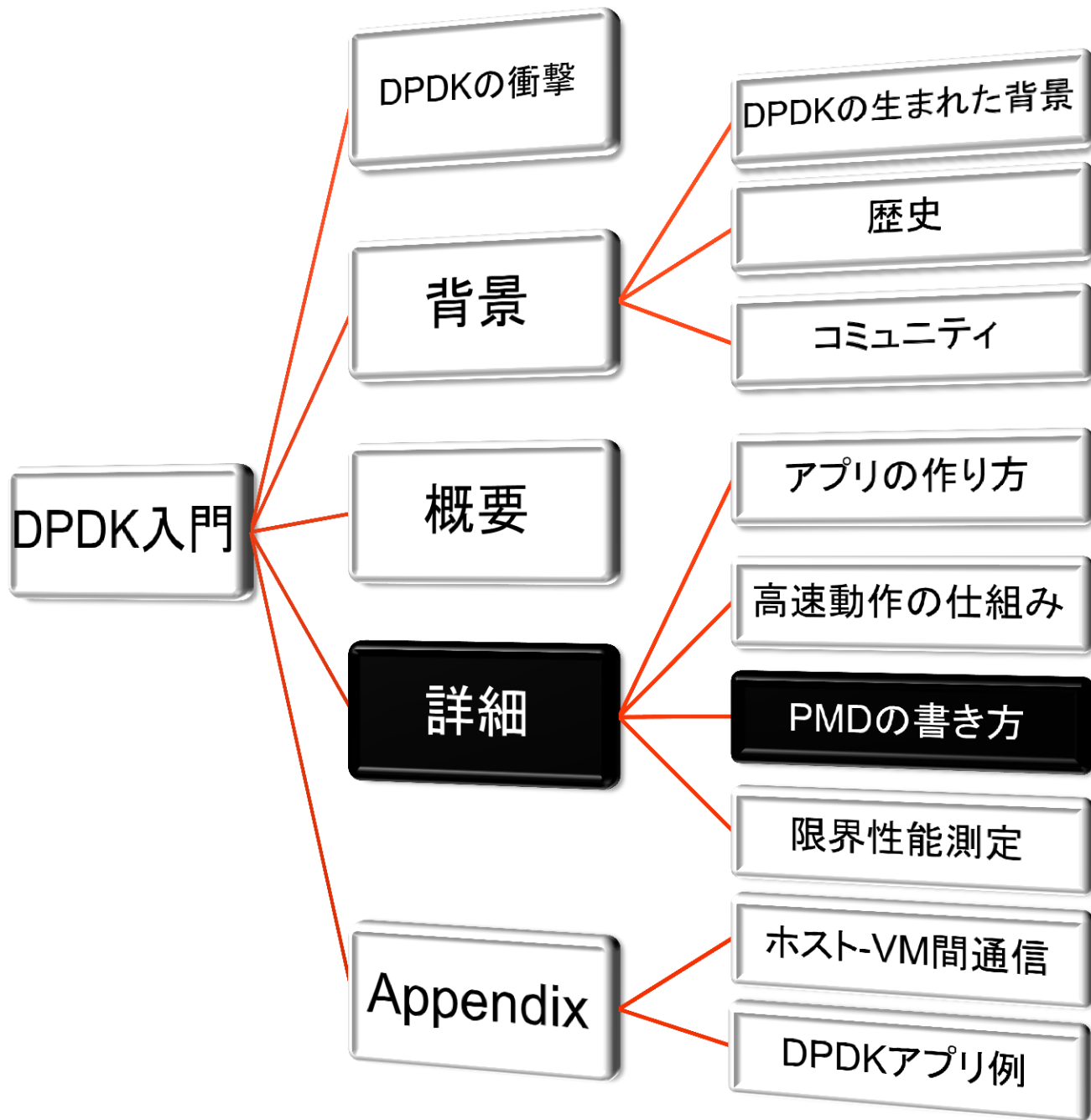


■ 1スレッドで1コアを占有する制約

- Ringは、スレッドがコアを占有していることが前提の実装
- 前述のように、RingはDPDKの根幹に食い込んでいるので、Ringを使えないのは致命的

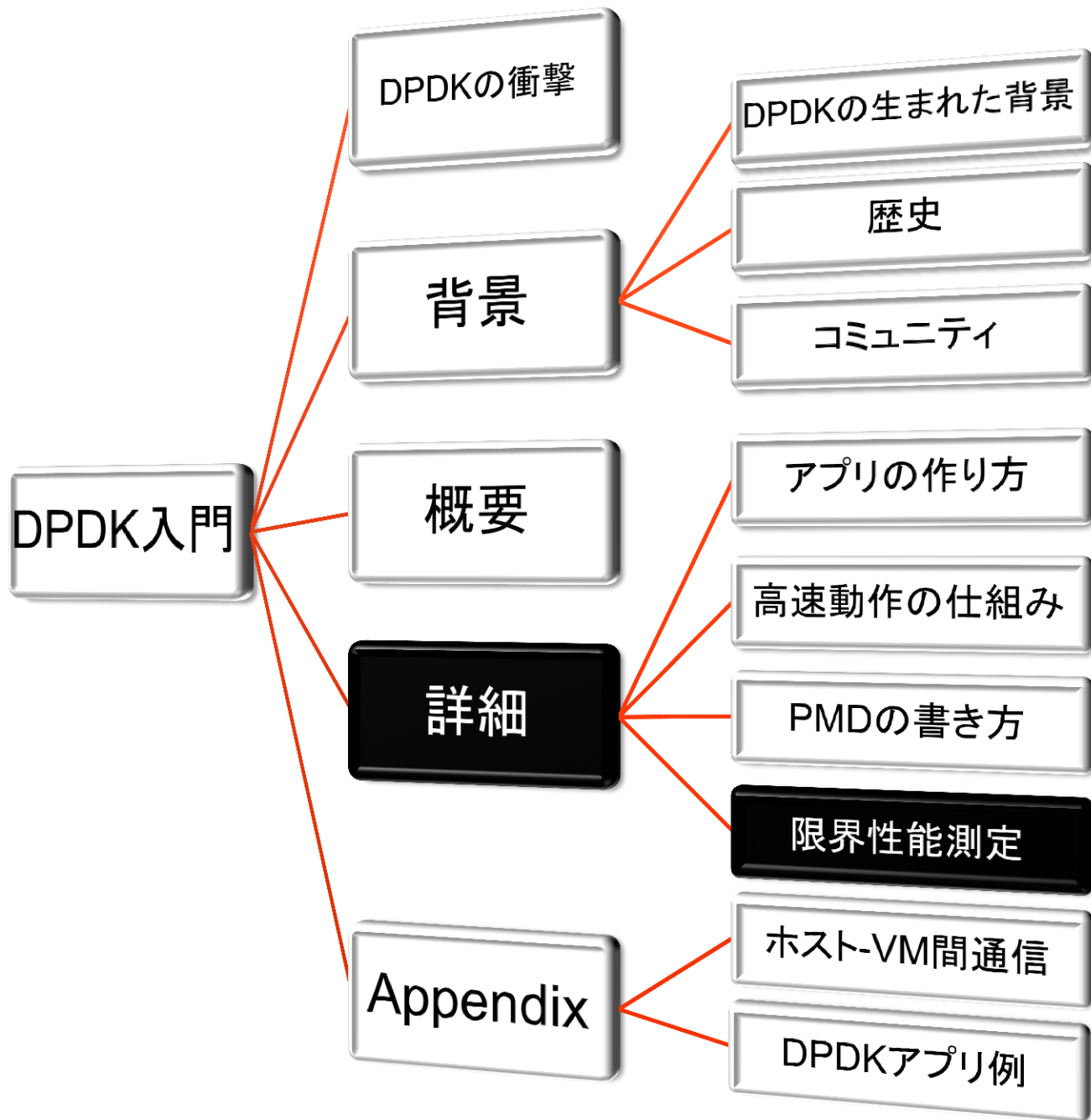
■ 同一ポートにアクセス可能なのは、1スレッドという制約

- PMDの実装による制約



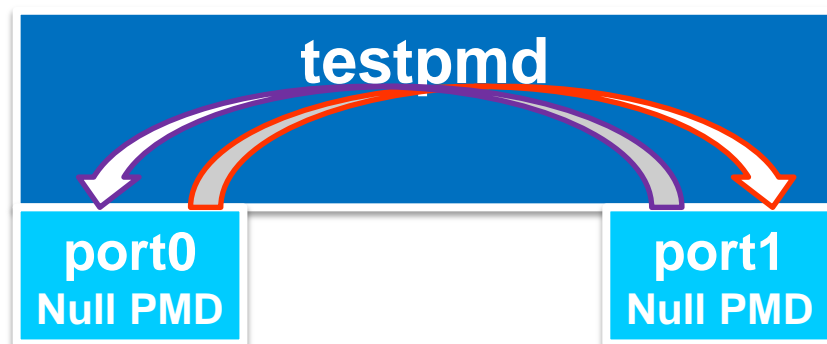
■ Null PMD

- /dev/nullライクなPMDを書いてみた。
 - 幾らでもパケットを受信できる。
 - 幾らでもパケットを送信できる。
- 仮想デバイスに対するPMD
- <http://dpdk.org/dev/patchwork/patch/686/>
- 非常に単純な構造なので、ひな型になるはず。
- Intel以外のNICでも、PMDを書けば、DPDKは動作するので、興味がある人は書いてみてください。



■ Null PMDを利用して、DPDKの限界性能を簡易測定

- testpmdは、2つのポート間の単純な転送を繰り返す。
- 2つのスレッドを持つ
 - port0⇒port1の転送を行うスレッド
 - port1⇒port0への転送を行うスレッド
- Null PMDは、(非常に小さいコストで)いくらでもパケットを送受信できるので、限界性能が簡易測定できる。
- 転送時に1コピー
 - 本当はゼロコピー環境が転送の限界性能に近いはずだが、今回は測定時間がなかったので、以前に測定した1コピー転送の結果



詳細

～限界性能測定②～

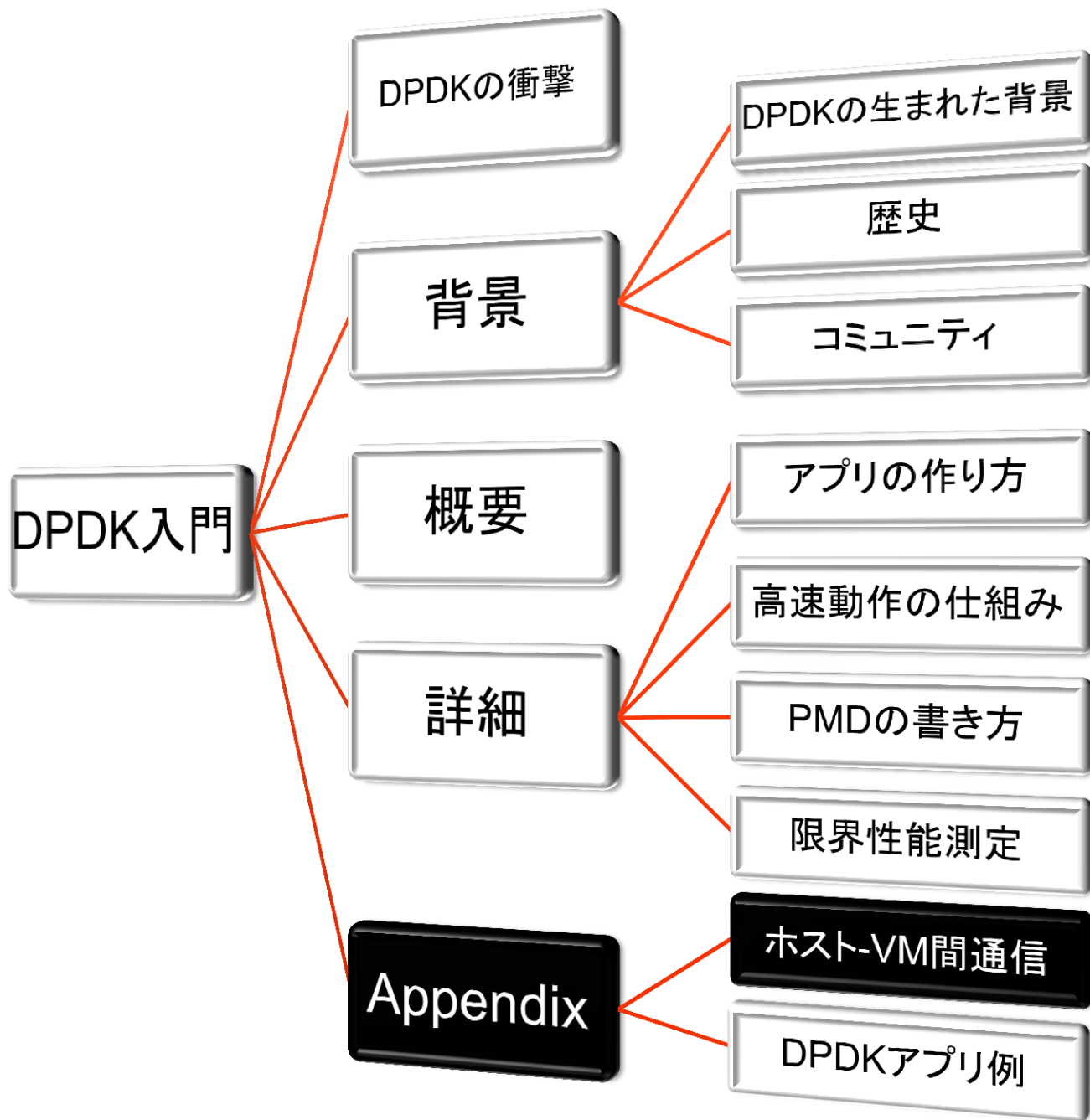
測定環境

CPU	Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
hugetlbfs割当	40GB (pagesize=1GB)

測定結果 ～コアあたりの転送性能～（かなり、ざっくり）

64byte	52Mpps / 26Gbps
1500byte	16.72Mpps / 195.05Gbps

1コピーでこの速度なら、ゼロコピーで160Mppsに届くのかも？
もしくは、複数コア/ポートを使って、160Mfpsという意味なのかも？
1コピーでこの性能なため、複雑なパケット処理を行うと、この値以下になりそう。



■ ホスト上のDPDKアプリと、VM上のDPDKアプリが高速に通信する仕組みについて

– なぜ、必要なのか？

- ネットワーク業界でよく言われる、SDN-NFVという構成を実現する際に必要になる。

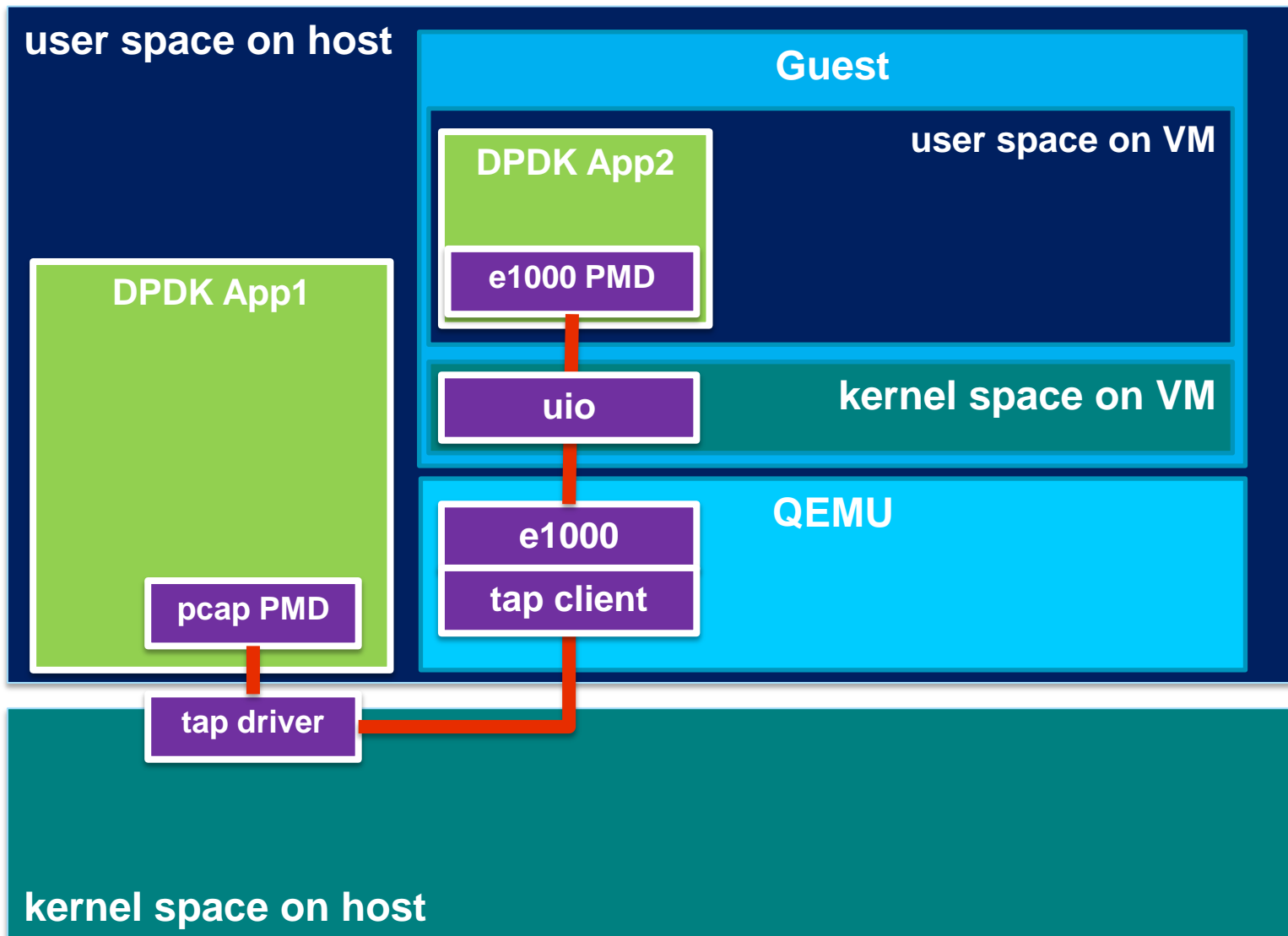
– SDN-NFVとは？

- ホスト上では、OpenFlowに対応したソフトウェアスイッチ(SSW)を動作させ、実際のパケット処理はVMで行うという構成

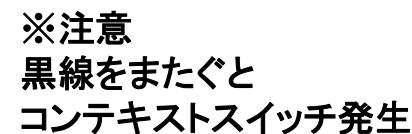
– LinuxとQEMU環境の場合について説明

ホスト-VM間通信

～e1000 & pcap経由～

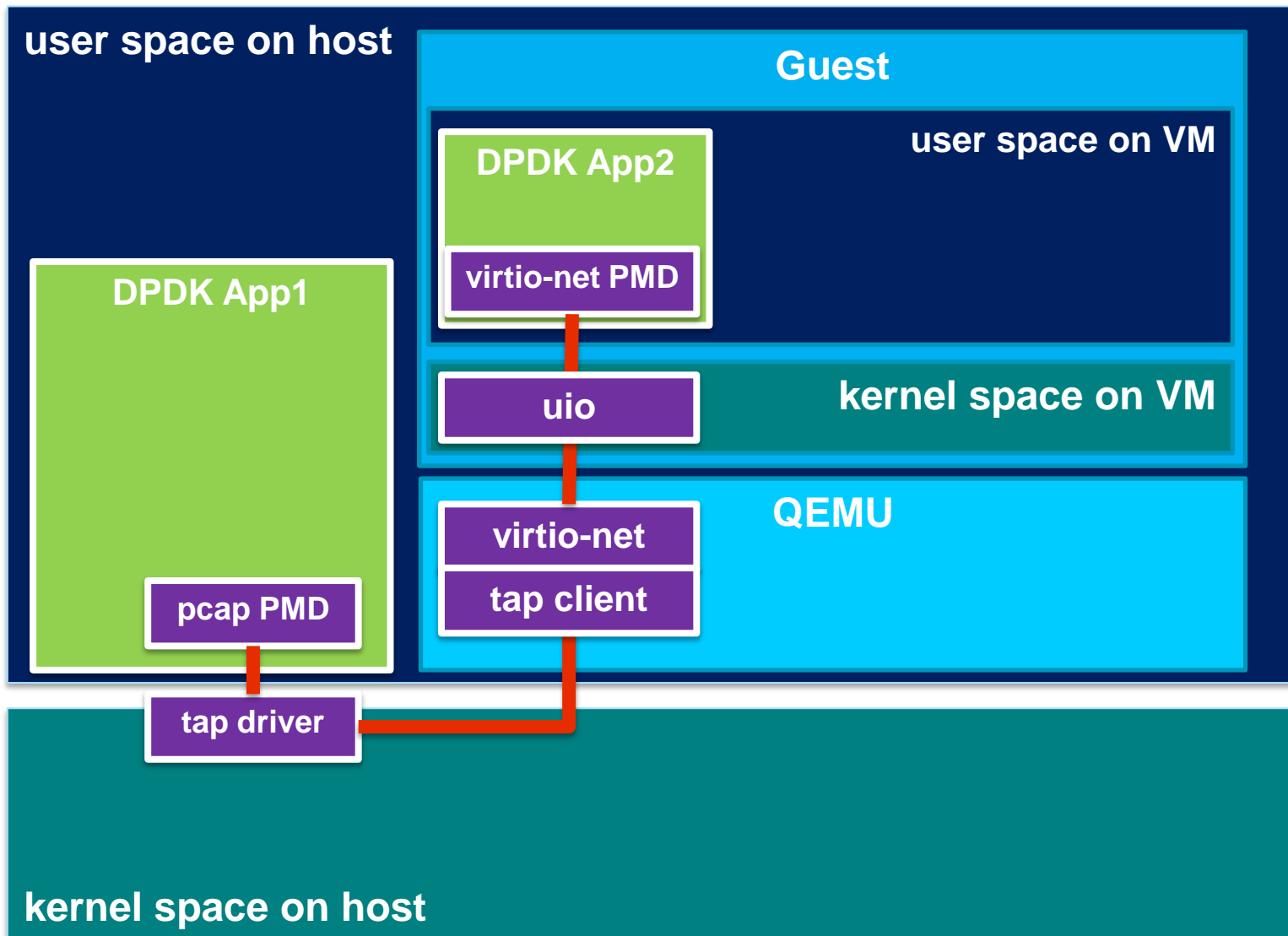


～e1000 & pcap経由～ App1からの送信



ホスト-VM間通信

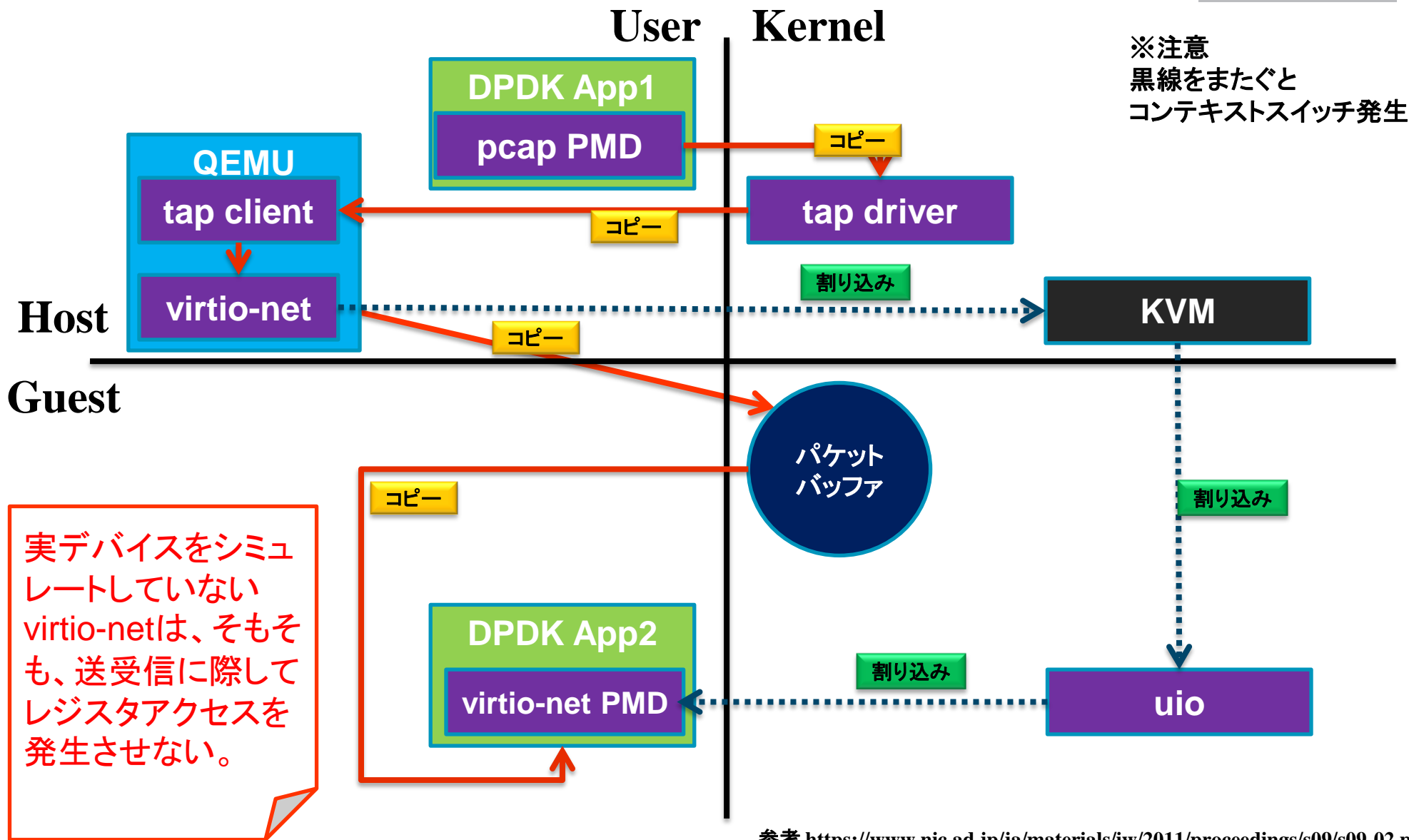
～virtio-net & pcap経由～



ホスト-VM間通信

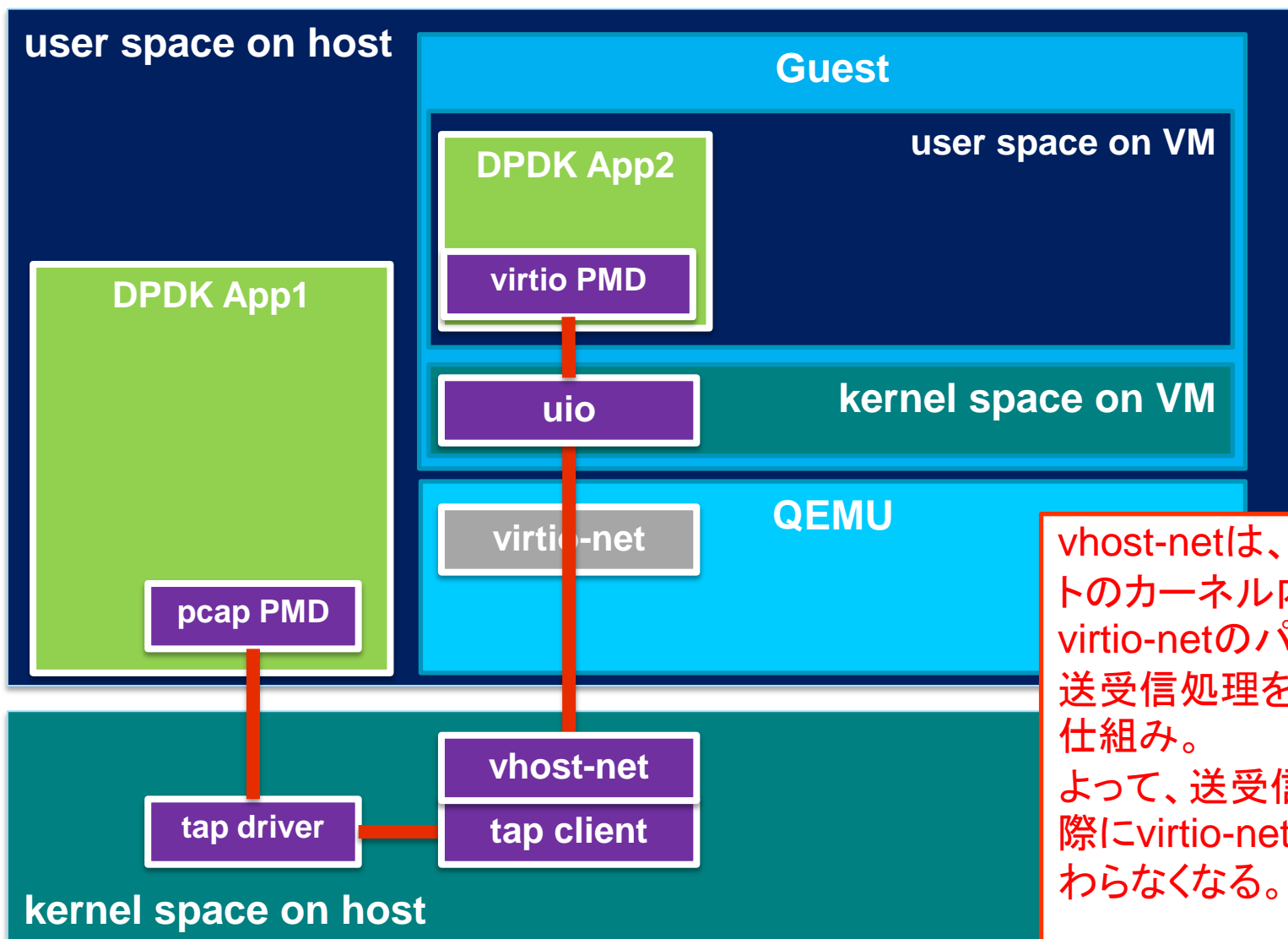
～virtio-net & pcap経由～ App1からの送信

igel



ホスト-VM間通信

～virtio-net & vhost & pcap経由～

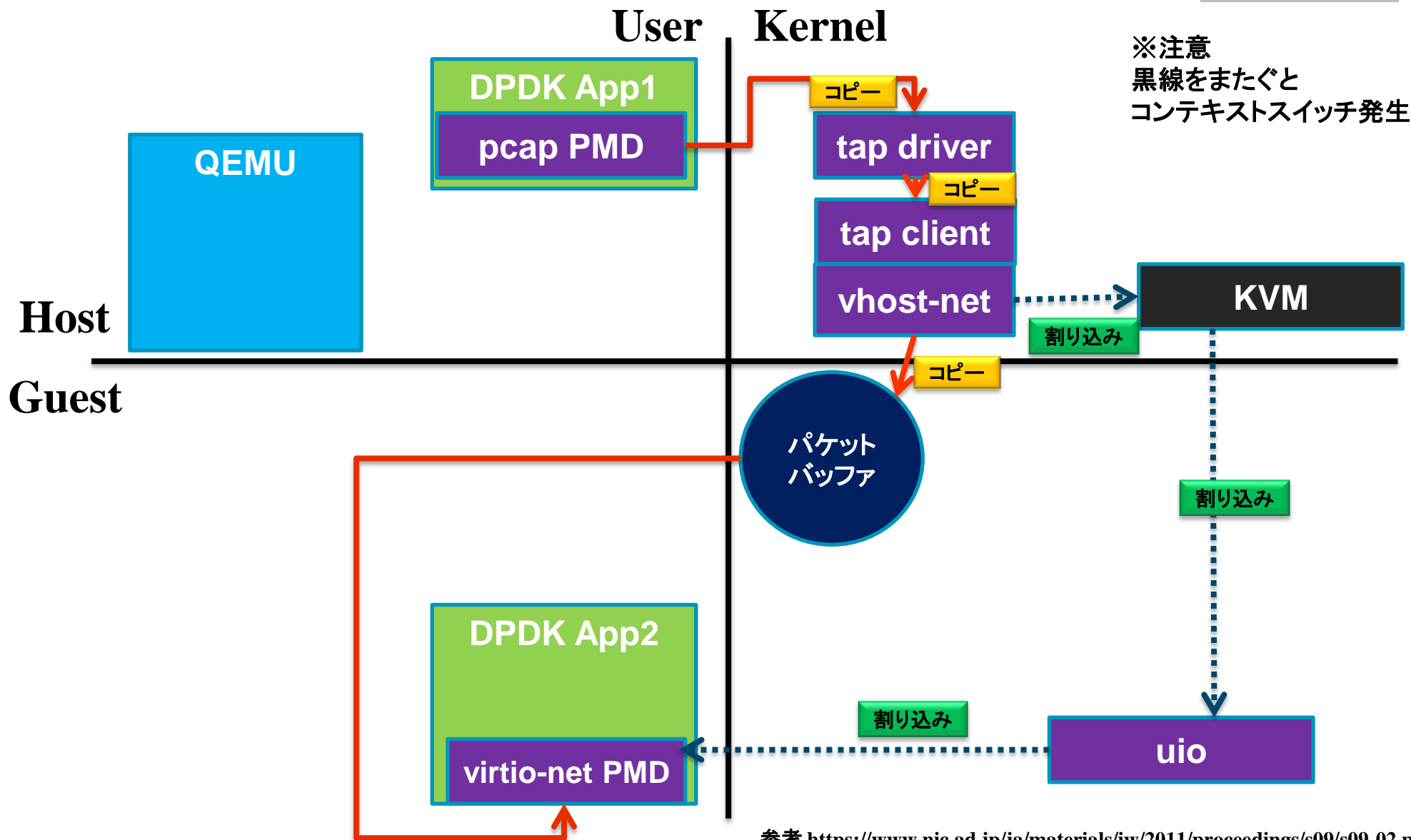


vhost-netは、ホストのカーネル内で、virtio-netの packet 送受信処理を行う仕組み。よって、送受信の際にvirtio-netは関わらなくなる。

ホスト-VM間通信

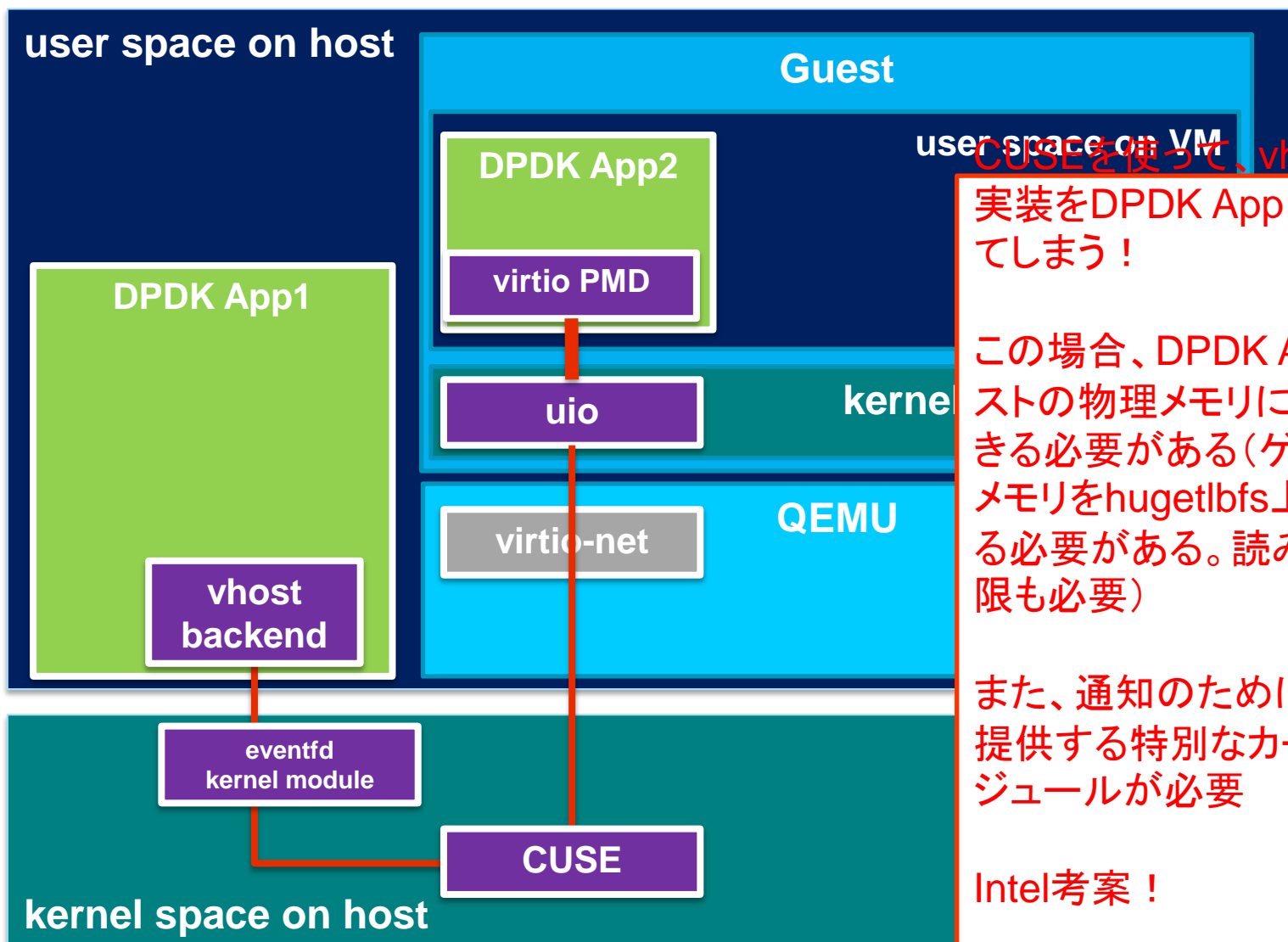
～virtio-net & vhost & pcap経由～ App1からの送信

igel



ホスト-VM間通信

～virtio-net & cuse & vhost backend経由～



CUSEを使って、vhost-netの実装をDPDK App1の中で行ってしまう！

この場合、DPDK App1からゲストの物理メモリにアクセスできる必要がある(ゲストの物理メモリをhugetlbfs上から取得する必要がある。読み書きの権限も必要)

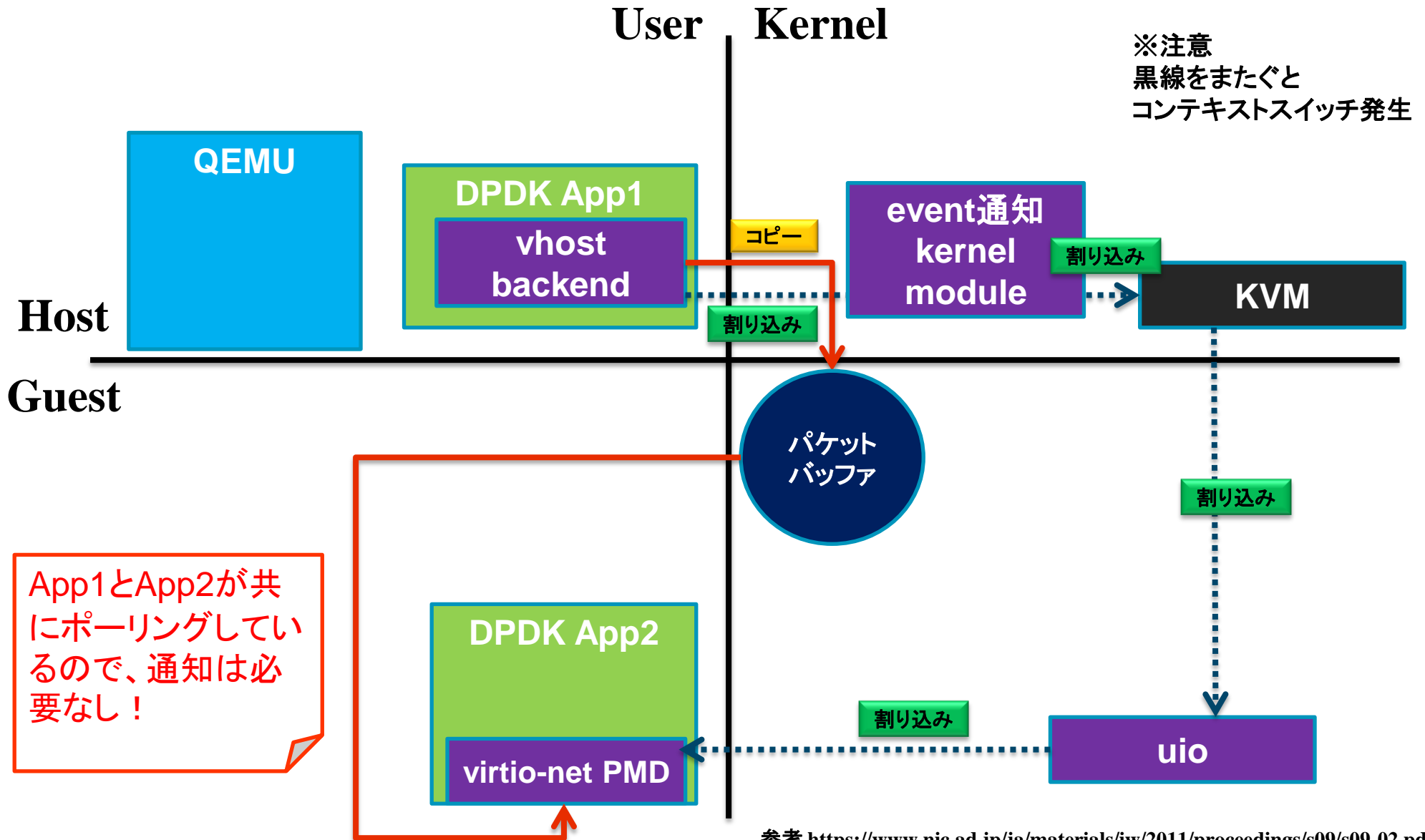
また、通知のために、DPDKの提供する特別なカーネルモジュールが必要

Intel考案！

ホスト-VM間通信

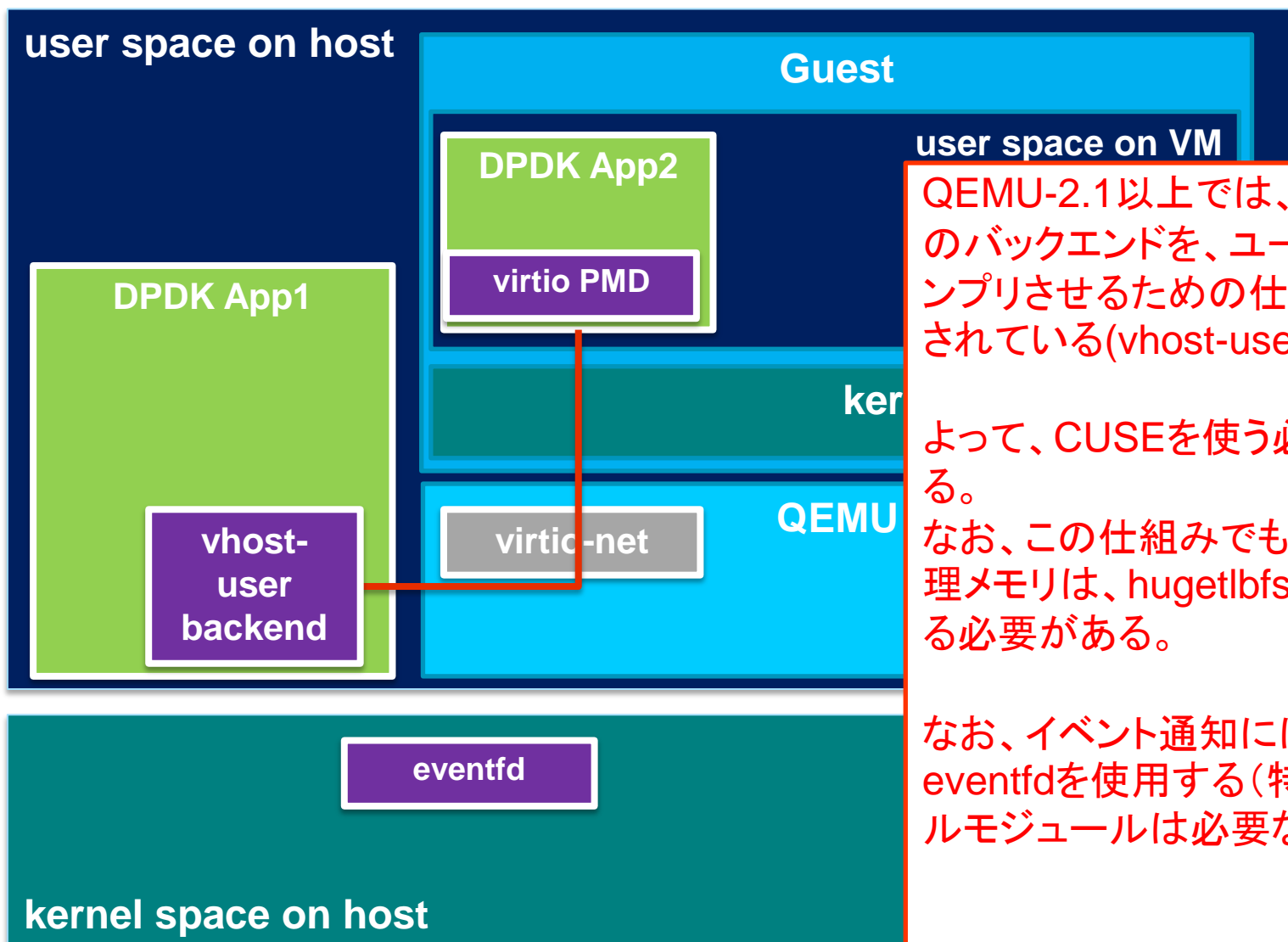
～virtio-net & cuse & vhost backend経由～ App1からの送信

igel



ホスト-VM間通信

～virtio-net & vhost-user backend経由～



QEMU-2.1以上では、vhost-netのバックエンドを、ユーザ空間にインプリさせるための仕組みが実装されている(vhost-user)。

よって、CUSEを使う必要はなくなる。

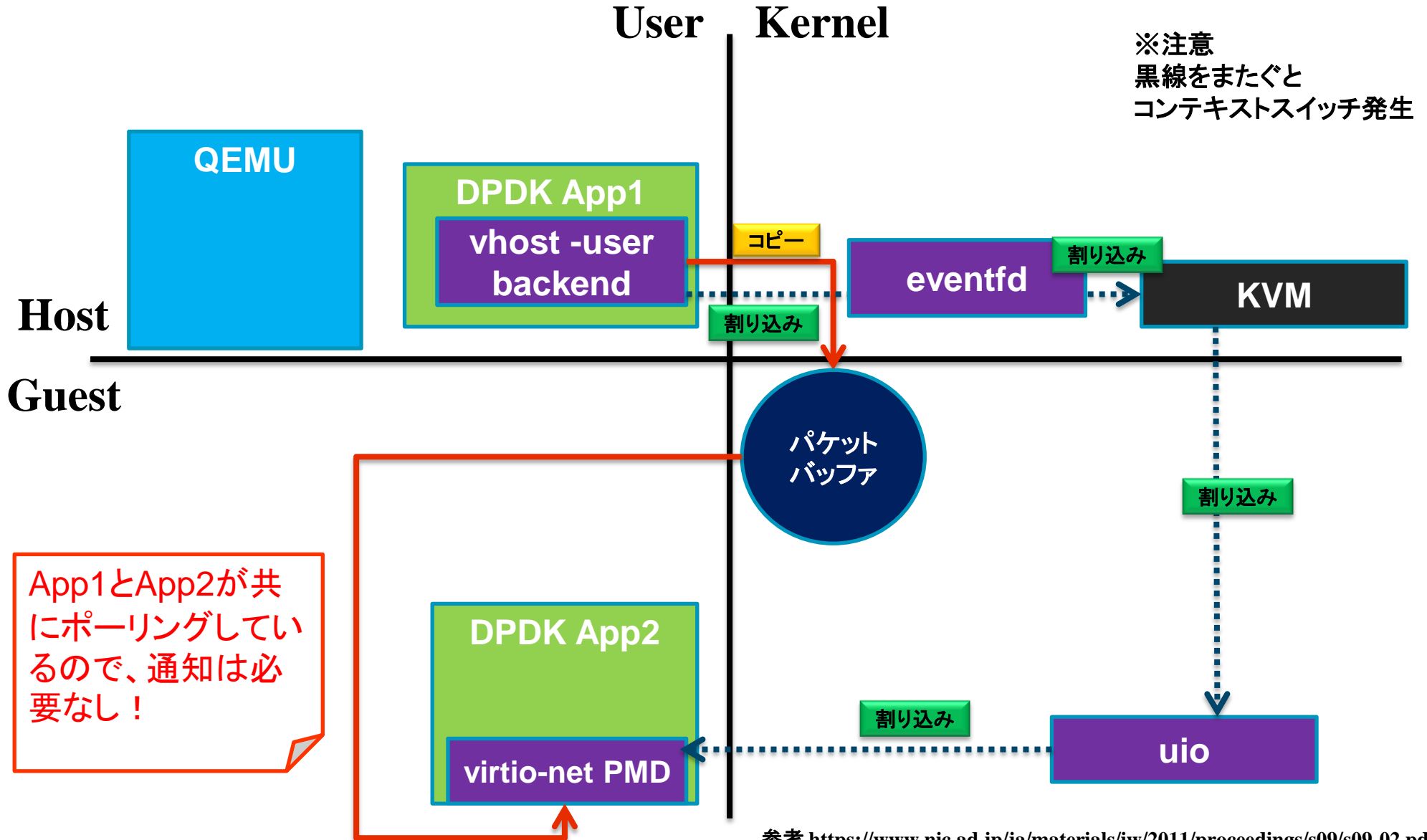
なお、この仕組みでも、ゲストの物理メモリは、hugetlbfsから取得する必要がある。

なお、イベント通知には、通常のeventfdを使用する(特別なカーネルモジュールは必要ない)。

ホスト-VM間通信

～virtio-net & vhost-user backend経由～ App1からの送信

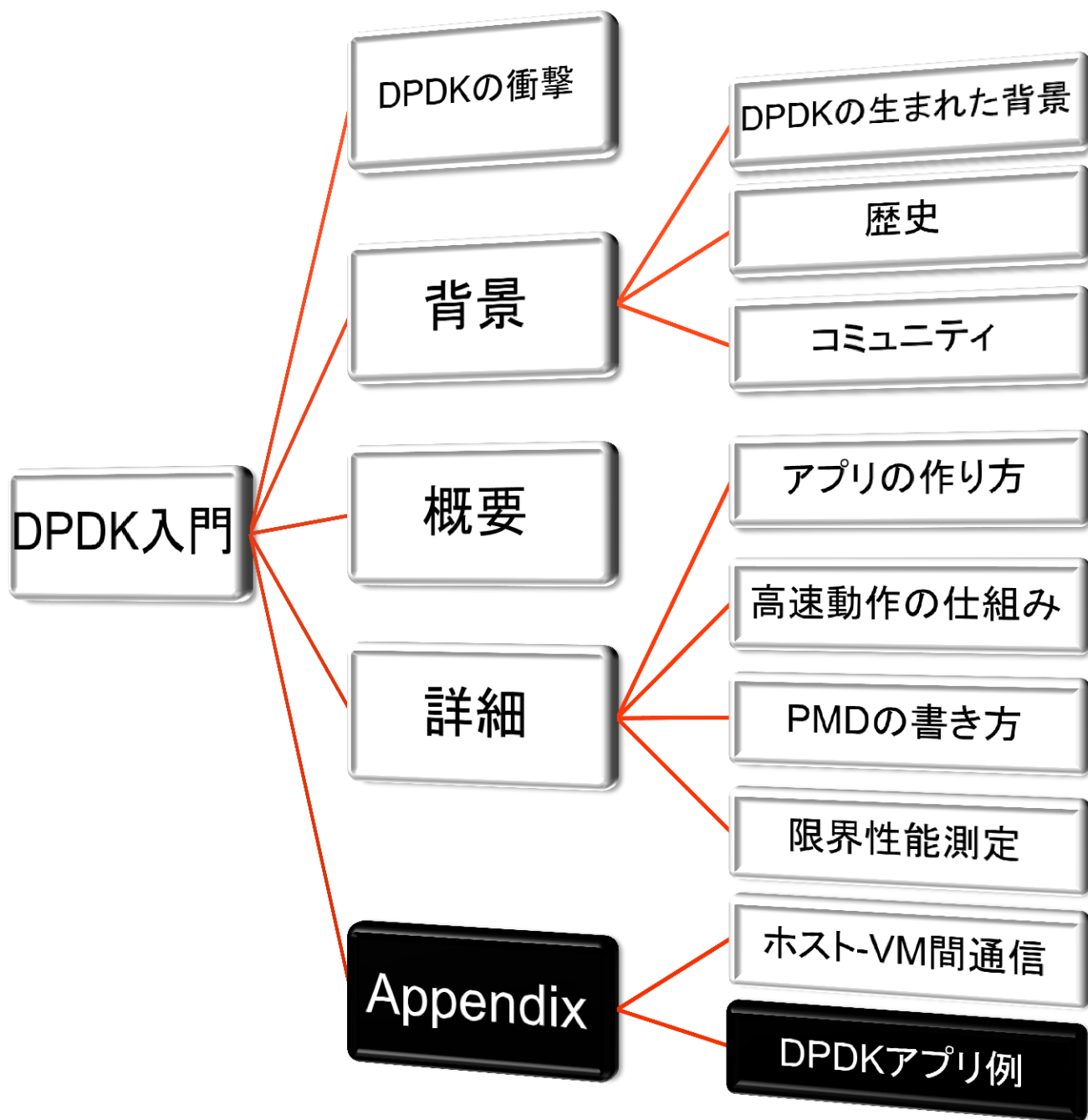
igel



ホスト-VM間通信

～その他～

- 他にも、いろいろありますが、ここでは割愛
 - IVSHMEM + Ring
 - MEMNIC PMD
 - NEC製
 - 高速な転送を実現するものは、何らかの共有メモリを実現して、そのうえでパケット交換するものです。



DPDKアプリ例

～open sourceのみ～

アプリ名	ライセンス	概要
lagopus	BSD	OpenFlow-1.3対応のソフトウェアスイッチ
dpdk-ovs	BSD	DPDK対応したOpen vSwitch
Packetgen	BSD	パケットジェネレータ
dpdk-rumptcpip	BSD	rumpkernelのDPDK対応
ipaugenblick	GPL/BSD	Linux TCP/IPスタックのDPDKポーティング
dpdk-odp	BSD	BSDのTCP/IPスタックをDPDKにポーティング (ソースコードをいつまでも公開しないので、使わない方が無難?)

- 既存のフレームワークでは、高速転送を実現できない
- DPDKは、ユーザランドから、CPU・メモリ・NICを扱う仕組み
 - NICは、uio/vfio経由
 - CPUは、Affinity管理
 - メモリは、hugetlbfs経由
- ポーリングを効率的に行うことで高速化を達成
- 実際に、DPDKを使用したアプリや製品が出始めている。
- DPDKのコミュニティ
 - Site: <http://dpdk.org/>
 - ML: dev@dpdk.org